TerraWeather
Ensuring Your Safety & Success

AI Workshop – Lecture 2

Running Caffe

1.0 Caffe Basics

Caffe trains your models in *phases*. For the time-series problems we encounter, this is done in two *phases*:

- The *training phase* is when your NN learns a task, given a *training dataset*. The Backpropagation (BP) learning algorithm (Lecture 3, tomorrow) is applied in this phase only.

- The *testing phase* is when your network's performance is reported for a separate *test dataset.* This performance lets you know how well your network is actually doing on new data. No learning is performed in this phase.

In the literature on NNs, you will see an additional “validation” phase used for categorisation tasks. We feel this is not suitable for time-series forecasting, since the chronological sequence of data matters and because you often don't have much data to work with.

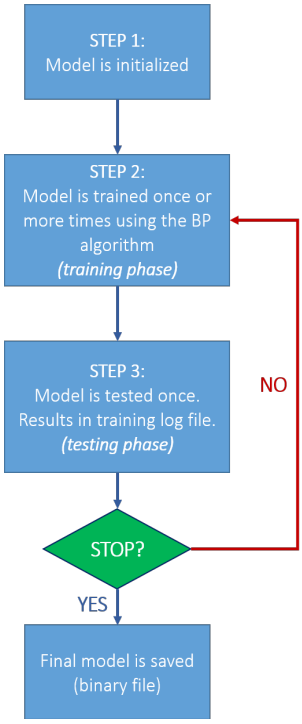

Figure 2.1: Overview of how Caffe trains a NN model.

From Figure 2.1, each batch of training phase is followed by a testing phase, during which the model's performance to date is reported. You know a model is trained when its performance metric (known as a *loss*) improves over each iteration of the main loop.

We'll discuss these phases in-depth tomorrow in Lecture 4. Both phases are loosely referred to as “training” your NN model.

1.1 Prototext

Caffe uses Google's **Prototext** format (.prototxt extension) to define your models. There are a few advantages of doing this, instead of using Python, as is the case with most other popular NN frameworks:

1. The prototext format is both human-readable (ie, textual) and can be converted to binary format (to save your trained models),
2. No programming is necessary to define new models. You just write out your model in prototext format,
3. The prototext file is converted internally into a C++ datastructure for control flow of your NN. This makes training and using your models faster than using Python for control flow.
4. You can train and load partial networks with prototext. This is very important for more complex training protocols (like SAEs, which we will investigate in Lecture 7 on Day 3).
5. The prototext format is easy to generate using other programs, which is helpful for automation (Lecture 8). This makes generating a wide number of network configurations easy.

You'll learn Caffe's prototext format over the upcoming lab sessions.

1.2 Caffe's Input Files

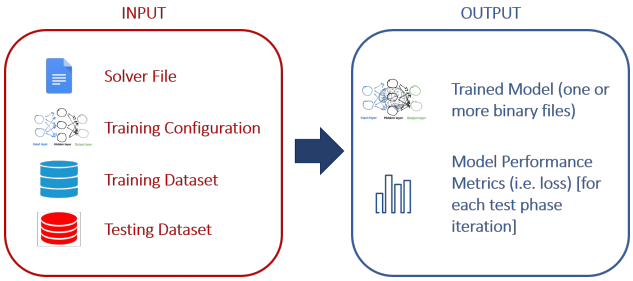

Figure 2.2 : Input & Output of Training / Testing Phases

There are 5 files you need to define an NN model in Caffe:

1. **solver.prototxt:** This prototext file defines the learning rate and other aspects of Caffe's "Solver". The Solver trains your neural network for a given task, using the Backpropagation algorithm for learning. Note that all frameworks need to utilize a solver.

2. **train.prototxt**: This prototext file defines your network and how to measure its performance. Both training and test phases must be defined in train.prototxt.

3. **test.prototxt**: Despite its name, **test.prototxt** has nothing to do with the testing phase. Instead, this is an optimized version of train.prototxt for you to run your models after they have been fully trained. We use test.prototxt to run predictions that will later be visualized.

4. Data files, **training.txt** and **test.txt** : these are text files containing links to the actual datasets used for train.prototxt and test.prototxt respectively. You can have more datafiles differently named, but we stick to these names by convention. The actual datasets are defined in an industry standard format called HDF5.

In today's "take-home" Lab, You will be provided with Python scripts, which you need to hack to convert textual timeseries data into HDF5 format. You don't need to know much about the HDF5 file format, but you **must** carefully study the Python sample scripts provided, since you will need this in the Datathon challenges.

1.3 Caffe's Output

During training/testing, Caffe outputs the trained model in binary format and a training log file.

- **train.log:** The training log is very valuable for diagnostics and debugging your NN model. In Lab 2A, you should take the time to study and familiarize yourself with the contents of the train.log file.

- **test.log**: This log file is produced after running the prediction. Use this to troubleshoot problems in the test.prototxt network.

- **binary model** (.model): The name of the model file would depend on the name you give it in the solver.prototxt file. Actually, you specify a name *prefix,* since Caffe will output more than one model, after every pre-

specified (also in solver.prototxt) training iterations. So, if you set the prefix to (say) “mynet_” and for Caffe to save models every 1000 iterations, during a 50,000 iteration run, you should have 50 model files, named “mynet_1000.caffemodel”, “mynet_2000.caffemodel” ... “mynet_50000.caffemodel”. Any one of these model files, together with a test.prototxt can be used to run your network on new input.

- **solver state** (.solverstate): This file is generated alongside the binary model files and is required to resume training the model.

1.4 Blobs, Dimensions and Predictions

Please study the material in this section carefully.

Suppose you want your network to predict temperature 3 days from the present, given humidity and rainfall up to the present time. Then:

- 3 days is your forecast *time horizon*. A forecast's time horizon is how far into the future a prediction is made.
- In this example, temperature is what we want to predict, and in Caffe, it is know as a *label*.
- Caffe's basic datastructure is called a blob. So, if the temperature is given for training every hourly over a 30 day period, we have 30 x 24 = 720 labels for temperature. These naturally fit into a 1-dimensional blob of length 720. Caffe calls this blob's dimension (720 1), meaning 720 values of 1 number (ie, temperature). It could also have dimension (720) or (720 1 1) which would all have 720 values.

The concept of a blob's dimension is central to Caffe, so be sure you understand it well. As a second example, if the input data is fed in as (humidity, rainfall) pairs hourly over 30 days, this is represented by a blob with dimension (720 2) ie, 720 values of 2 numbers.

During training, a Caffe model simply transforms an input blob into an output blob, then compares the output with the labels. The output blob must be the same dimension as the label blob.

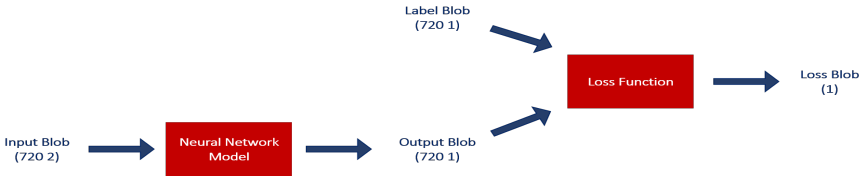

Figure 2.3 – The blob transformations in the example model.

In Figure 2.3, the Output and Label blobs are compared element-wise using a Loss function. The result is a loss blob of dimension (1). For example, if the loss function were the mean absolute error ($loss_i = |label_i - output_i|$), then the average loss is the sum of $loss_i$ where i = 1 to 720, divided by 720. This is just a single number, so its dimension is (1). This number is reported in train.log at every training iteration.

1.5 Creating new HDF5 Input Files

The HDF5 file format can store many large datasets organised in a hierarchy. For our purposes, we will just use it to store two datasets – one containing the input variables that will be fed to the NN, and the other the labels.

We use the **`h5py`** python package to create and read HDF5 files. This gives us easy access to the extensive math libraries in python which can be used for data pre-processing and preparing the input dataset. Pre-processing the raw data includes things like filtering off invalid values, dealing with missing values, aggregating or transforming the raw inputs. This is an important step as real-world data often contain errors or inconsistencies that reduces the quality of the training data and can impair the training of the NN.

1.6 Using Autocaffe to Schedule Tasks

There is no GUI (Graphical User Interface) for Caffe. You have to run it using a Bash script from the Terminal.

The alternative is to use **Autocaffe**, a tool we developed at Terra Weather for our own work, which makes it very easy to train/test your NNs. Autocaffe's most basic feature is *job scheduling*. A “job” is a NN train/test cycle. Using Autocaffe's job scheduling means you don't need to wait for your server to become free to run the next job. Autocaffe does it for you.

There are two important sub-folders in your Autocaffe installation:

1. **`experiments`** is where you should save your Caffe Input files.
2. **`results`** is where your trained models, log files and output will be saved.

You should create a sub-folder with each new NN model you wish to train in the experiments folder. For example, if you have an NN called "**`bingo`**" then you should create the sub-folder **`experiments/bingo`** and save your Caffe input files there.

Once you have done this, you need to "prep" your NN using the command:

```
ac prep bingo
```

This copies your input files into a corresponding folder in the **`results`** sub-folder. All your output will be saved here too. To start training, you must submit your NN to Autocaffe using the "add" command:

```
ac add bingo
```

This will schedule the job as soon as possible on the server. You can submit as many jobs as you wish, Autocaffe will schedule them as soon as the server becomes available. To check the status of the job queue, you use the "q" command:

```
ac q
```

This will output all *pending* (ie, scheduled but not yet running) jobs on the server. Use the "list" command to see all jobs currently being run:

```
ac list
```

To remove a job from the queue, use the "del" command:

```
ac del bingo
```

To stop jobs currently being run, you use the "stop" command:

```
ac stop bingo
```

You can prep/del/stop multiple jobs using the wildcard (*). eg:

```
ac del bing*
```

Will remove all pending jobs matching **`bing*`**

We will use Autocaffe extensively in this workshop, so you should familiarize

yourself with **ac** commands.

1.7 Visualizing your Results

In theory, the loss function is supposed to measure how well your model predicts the future. However, especially for time-series predictions, the loss does not always give you the full picture of how well your predictions are actually doing. You will see examples of this enough during the Labs!

This is because there is usually a disparity between what we gauge to be a "good" prediction and what the loss function *actually* measures. This is an important observation that you must remember.

The best way to gauge a model's performance is to simply plot out the actual vs predicted values for both the training and test datasets. Also of use is the change of the loss function during each iteration of the training and test phases. If you know how to interpret them, these graphs can be very helpful in tuning your model. We'll teach you how in Lecture 4 tomorrow.

Caffe focuses on training NNs, so it does not have builtin visualization tools. It is really up to you to write Python scripts to plot these graphs.

Besides job scheduling, Autocaffe also has built-in visualization. To use it, **after** your job has completed, you need to run the "test" command to collect the predictions over the test dataset:

```
ac test bingo
```

This creates a "**predictions.csv**" file in the **results/bingo** folder by running the NN specified by `test.prototxt` and extracting the predictions from the log files. You can import this CSV file into Excel if you wish to visualize the predictions. However, you can use Latex (Latex is a widely-used program for scientific reporting), which will be covered in Lab 2A today. To create a PDF report, you use the "report" command:

```
ac report bingo
```

This creates a PDF report of all the figures you have using Latex. The results are saved into Autocaffe's **reports** folder. You'll see this in action in Lab 2A.

Lab 2A: Running a toy example: Sine wave prediction.

The purpose of this Lab is to:

- Walk you through creating prototext for the Solver / Training / Test files,
- Show you how to train your NN using Autocaffe,
- Visualize the results using Excel,
- Printing PDF reports using Latex and Autocaffe.

Instructions

Sine wave prediction
This is a simple problem for you to try setting up a NN, training it and then analysing the results. The aim is to train a NN to predict the next value in a sine wave given the previous 5 values. To do this the sine function is sampled at regular intervals over one cycle, ie $\sin(N \cdot x), x = 2\pi/305, N = 1,2,\ldots$.

Training and test examples are made by taking 6 consecutive samples of the sine wave, eg $N = 3,4,5,6,7,8$, and using the first 5 values as the NN input and the 6th value as the label. In this way, 300 windows of 6 values (305-6+1) can be made. The first 200 windows are used for training while the remaining 100 are used for testing the model. Thus the training and testing datasets have a dimension of `(200 5)` and `(100 1)` respectively.

Set up
As mentioned in the notes, there are five files that together describe completely the composition and operation of a NN. These files are written in plain text, so we use the Vim text editor in this section.

Set up a folder to contain all the files related to this experiment by entering into the VM command line

```
mkdir ~/caffe/experiments/sine
```

and then cd into that directory.

train.txt / test.txt
We start with defining the train and test dataset location. These datasets contain the NN inputs and variables which are used during the NN training process and for the testing the NN after the training process is completed respectively.

The datasets have already been downloaded into the VM and are located in

the directory `~/caffe/data/sine` as `train.h5` and `test.h5` files. Check that these files exist in your VM. We let Caffe know of their location through the `train.txt` and `test.txt` files. Create a new file name `train.txt` with Vim and type in the path

```
/home/terra/caffe/data/sine/train.h5
```

Save and quit Vim. Now create `test.txt` but this time type in two paths

```
/home/terra/caffe/data/sine/train.h5
/home/terra/caffe/data/sine/test.h5
```

Since two files as specified, Caffe will read both datasets and combine them into one, in the order in which they are specified, ie `train.h5`, then `test.h5`.

Caffe layers

NNs are constructed in terms of "**layers**" in Caffe. A layer is an object that takes in data in the form of "**blobs**", performs some operation on them, and then outputs the result of the operation through one or more blobs. Caffe offers more than 50 different layer types, performing commonly-used operations ranging from slicing (splitting) a blob into multiple parts to applying an inner-product between a matrix that is stored internally in the layer and a blob. You will use these layers to build up your networks.

The general format for specifying a layer is given below:

```
layer {
     name: <layer name>
     type: <layer type>
     bottom: <bottom blob 1>
     bottom: <bottom blob 2>
     top: <top blob 1>
     top: <top blob 2>
     <layer type>_param {
          parameter1: 50
          parameter2: true
          …
     }
}
```

- **Layer name**: Each layer must have a name. It can be anything as long as it's unique within the network
- **Layer type**: This determines what operation the layer will perform. Details on the built-in Caffe layer types can be found here
- **Layer parameters**: This is used for layers-specific parameters that

affect layer operation
- **Bottom blob(s)**: These are the blobs containing the input data that will be operated on by the layer
- **Top blob(s)**: These are the output blobs to which the layer writes the results of its operation

Note: These layer properties can be written in any order in the prototext file – you don't have to follow the order shown in the examples here. However the order of the layers themselves are important, in particular layers with bottom blobs cannot be defined before the layers producing those blobs (ie the layers whose top blobs are used as bottom blobs in the layer in question).

train.prototxt

We are now ready to start creating our own networks. The layer definitions are written in the `train.prototxt` and `test.prototxt` files. Let's start with the `train.prototxt`, which defines the NN used in the training phase.

Create the `train.prototxt` file in Vim and put in the first line the name of the network

```
name: "Sine"
```

This name is just an identifier for the network, but do remember to add this line as Caffe will produce errors without it.

As the sine wave prediction problem is a relatively easy for NNs to learn, we will use a simple 4 layer network. The first layer is an input layer that reads a HDF5 file and outputs the datasets stored in that file as blobs. Refer to the layer definition below:

```
layer {
     name: "input"
     type: "HDF5Data"
     hdf5_data_param {
          batch_size: 20
          source: "train.txt"
     }
     top: "data"
     top: "label"
     include: {
              phase: TRAIN
     }
}
```

Unlike the general layer format earlier, the HDF5Data layer does not have

bottom blobs as its inputs are read from the HDF5 files directly. The source file is given in the layer parameters section and the bottom blobs give the name of the datasets (stored in the source file) that are output as blobs.

In the `train.hdf5` dataset for this problem, the "`data`" dataset is the input to the NN and consists of 200 examples of 5 consecutive values of a sine wave. The "`label`" dataset contains the "answer" that we want the NN to produce – ie the value following the 5 input values along the sine wave for the corresponding 200 examples.

When the HDF5Data layer writes the top blobs on each training iteration, it reads a fixed number of examples, set by the `batch_size` parameter. In this case the batch size is 20, so the layer reads 20 examples at each iteration, moving on to the next set of 20 examples on the next iteration. After 10 iterations, the entire training set has been output to the NN and is known as an **epoch**. After one epoch, the layer loops back to the start of the dataset and this repeats until the end of the training process.

As described in Section 1.0 of the lecture notes, the training process consists of two phases – the train and test phase, each with a different dataset to feed to the NN. As such, we require two HDF5Data layers, one to be used for each phase. Caffe makes it easy to do this with the `include` layer parameter. As you can see, the layer given above is for the training phase. For the test phase, we have a nearly identical layer

```
layer {
     name: "input"
     type: "HDF5Data"
     hdf5_data_param {
          batch_size: 20
          source: "test.txt"
     }
     top: "data"
     top: "label"
     include: {
              phase: TEST
     }
}
```

The only difference between the two are the `source` file in the HDF5Data layer parameter and the `phase` setting under the `include` parameter.

The next two layers are inner product layers, which take the input data and produces a prediction value. More details on this layer will be covered in the

layer lectures.

```
layer {
     name: "inner-product"
     bottom: "data"
     top: "ip"
     type: "InnerProduct"
     inner_product_param {
          num_output: 4
          axis: 1
          weight_filler {
               type: "xavier"
          }
          bias_filler {
               type: "constant"
               value: 0
          }
     }
}

layer {
     name: "output"
     bottom: "ip"
     top: "prediction"
     type: "InnerProduct"
     inner_product_param {
          num_output: 1
          axis: 1
          weight_filler {
               type: "xavier"
          }
          bias_filler {
               type: "constant"
               value: 0
          }
     }
}
```

The loss layer applies the loss function, in this case the Euclidean loss function of $(prediction-label)^2$, on the prediction and label blobs to produce a loss value.

```
layer {
     name: "loss"
     bottom: "prediction"
     bottom: "label"
     top: "loss"
     type: "EuclideanLoss"
}
```

test.prototxt
To define the test network, create the `test.prototxt` file, which requires the following changes, but is otherwise identical to the training network in `training.prototxt` (tip: you can copy the `train.prototxt` file using the `cp` command to save some typing)

1. (Optional) Omit the training-phase HDF5Data layer
2. Omit the entire Euclidean loss layer

The first change is optional since the test process runs with the phase `TEST` by default. The loss layer can be omitted as no training is done when testing the model, hence the loss value is not required.

The test process is distinct from the test phase during the training process. Both use the same input dataset, but whereas the test phase is invoked periodically during training to check on the progress of the partially-trained model, the test process is performed after the model is trained to see the performance of the fully-trained model.

solver.prototxt
This file contains a list of parameters related to **training** of the NN. Copy the following settings, one per line onto your solver.prototxt file.

```
net: "train.prototxt"
```

This specifies the name of the prototxt file containing the NN structure used for training.

```
  base_lr: 0.001
lr_policy: "fixed"
```

These specify how the learning rate changes throughout the training process. In this example, the learning rate is fixed at 0.001 and does not change. The learning rate and learning rate policies will be explored in future lectures.

```
test_iter: 15
```

The `test_iter` sets the number of iterations to run during the test phase. As the size of the test dataset is 300 and the batch size of the input layer is 20, 300/20=15 iterations are needed for the NN to consume the entire test dataset.

```
test_interval: 50 # every 5 epochs
   display: 50 # every 5 epochs
```

`test_interval` sets the number of training phase interations to run before a test phase is triggered, while `display` sets the number of training phase iterations before the output of the network (eg. loss) is displayed for logging. Setting these two values the same means that each time the output of the train and test phases will be printed together on the same iteration.

```
max_iter: 2000
```

This line sets the end-point for the training process – training stops after the specified number of iterations is reached.

```
snapshot: 2000
snapshot_prefix: "sine"
```

As described in Section 1.3, snapshots capture the state of the model. We just want to get the final trained model, so we set the snapshot setting equal to the `max_iter` value of 2000.

```
solver_mode: CPU
```

This setting lets you choose whether to run Caffe on the CPU (Central Processing Unit) or the GPU (Graphics Processing Unit). Since we are running Caffe in the VM, please set this to CPU for all experiments.

Training the network

Now that we are finally done with specifying the network, let's proceed on to training it. You should have 5 files in the `~/caffe/experiments/sine` directory. We use Autocaffe to schedule and train the NN, so refer to Section 1.6 and 1.7 for the commands.

Step 1: Enter `ac prep sine`
Step 2: Enter `ac add sine/1`
Step 3: Check that the job in on the queue with `ac q`
Step 4: Wait for the job to get scheduled and finish running. You can monitor the status of running jobs using `ac list`. Verify that the job has completed by checking that both the job queue and list are empty.

Step 5: Go to the `~/caffe/results/sine/1` folder. In addition to the 5 original files in the folder, there should be 3 more files – `train.log`, `sine_iter_2000.caffemodel` and `sine_iter_2000.solverstate` – if the training process was successful. If you do not see those files, you will need to troubleshoot the problem (see below). Looking into the `train.log` file may be useful in finding the error.

Step 6: Run the predictions with `ac test sine/1`. This command produces a `predictions.csv` containing the predicted values and labels which can be easily read by any spreadsheet or data analysis software. If the `predictions.csv` file is missing or empty, there is an error in the `test.prototxt` file. Check also the `test.log` file to see the error messages.

Troubleshooting: If your NN training fails, do not despair. As there is a lot of typing involved in setting up the prototext files, there is probably a typo hidden somewhere. Open the `train.log` or `test.log` file and look for the line

```
*** Check failure stack trace: ***
```

near the end of the file. The log messages before that line are the last operations performed by Caffe before encountering the error, and often give some description of what the issue is.

After you have found and fixed the issue, remember to do an `ac prep` command so that the updated files in the experiments folder is copied over to the results folder for training.

Visualization

Visualizing your results is extremely important, not only to get a better sense of the results and possibly gain some insight, but also to share your results in a clear and accessible form. Making a plot from data in csv format is a common operation, you may use your favourite tool like gnuplot or python notebooks if you prefer. For this exercise we will be using Excel spreadsheet as our tool or choice.

The goal for this section is to create two plots:

1) Plot of predicted values and labels over the test dataset
2) Plot of the training loss and test loss against iteration number during training

To do this, you have to transfer the `predictions.csv`, `train.log.train` and `train.log.test` files from the results folder (`~/caffe/results/sine`) to your host machine (refer to the Lab 1.2).

Plot 1

Open a blank workbook as well as the `predictions.csv` file in Excel. Copy columns A (label) and B (prediction) from the `predictions.csv` file to the blank workbook. Then plot both columns by selecting them and inserting a

line chart (see Figure L2A.1)

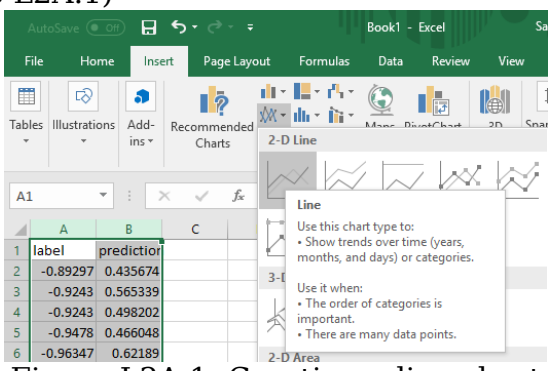

Figure L2A.1: Creating a line chart

The plot provides much information about the prediction accuracy of the NN model and is very useful in time-series prediction for getting a better picture of the model behavior than can be inferred from the loss alone.

Plot 2

This plot uses the data in `train.log.train` and `train.log.test`. These two files, despite their extensions, are actually also csv files and contain the train and test phase losses recorded during then NN training. Open them in Excel and select the entire column A. Then go to the Data tab and click on the 'Text to Columns' button (see Figure L2A.2 below).

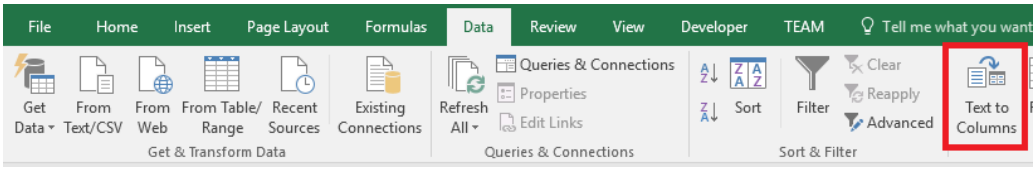

Figure L2A.2: Select the 'Text to Column' button

In the 'Convert Text to Column Wizard' dialog that pops up, choose the 'Delimited' option and click next. In the next set of options, look out for the checkbox labeled 'Comma' in the 'Delimiters' section and make sure that it is **checked**. Then click on the 'Finish' button. Repeat the above steps for both files.

You should end up with 4 columns in both `train.log.train` and `train.log.test`. We are interested in the **NumIters** column, which gives the training iteration number for that row, and the **loss** column. Create a new sheet in your Excel workbooks and copy both the NumIters and loss columns from `train.log.test` and the loss column from `train.log.train` into them. We only need one copy of the NumIters column since it should be the

same in both files due to the display and test_interval settings in `solver.prototxt` being set identical (apart from the final row in `train.log.test`).

Finally, rename the header of the loss columns copied from `train.log.train` and from `train.log.test` to 'train loss' and 'test loss' respectively. Plot the graph by selecting the three columns and inserting a scatter plot.

This graph gives you much information about how well the training process is going. For example if the training loss decreases with iteration number while the test loss is increasing, this is a sign of over-fitting, where the NN is learning a relation specific to the training dataset and does not generalise to the test set. More on this will be covered in future lectures.

Autocaffe

You may have noticed a `figs` folder in the results folder of the sine experiment. This folder contains the plots generated automatically when you ran the `ac test` command. In the `figs` folder, there should be the following files:

- `loss-training.png`: Plot 1 of the previous section
- `prediction.png`: Plot 2 of the previous section
- `test_prediction.png`: Same as `prediction.png`, but with the view limited to the predictions over the test dataset

Generating reports

Autocaffe can generate a report for experiments in the results folder. The format of the report is specified using Latex in the `report.tex` file that is read from the results folder for the experiment (`~/caffe/results/sine/1/`). Latex is a type-setting system that can produce publication-quality documents. You can learn more about it here.

Create a `report.tex` file with Vim in the experiments folder (`~/caffe/experiments/sine/`) and copy the Latex code below:

```
\section{Sine curve prediction}
```

This creates a new section named "Sine curve prediction".

```
\begin{figure}[H]
	\centering
```

```
       \includegraphics[width=\textwidth]
{/home/terra/caffe/results/sine/1/figs/prediction.png}
       \caption{Prediction}
\end{figure}
```

This is the format for insert a figure. The `[H]` is used to force the placement the figure to where it is written in the code (Latex may shift the position of figures to optimize the page usage). `\centering` tells Latex to align the figure at the center. The purpose of the other commands can be inferred from their names.

```
\begin{figure}[H]
    \centering
       \includegraphics[width=\textwidth]
{/home/terra/caffe/results/sine/1/figs/test_prediction.png}
       \caption{Prediction (test set)}
\end{figure}

\newpage

\begin{figure}[H]
    \centering
       \includegraphics[width=\textwidth]
{/home/terra/caffe/results/sine/1/figs/loss-training.png}
       \caption{Training Loss}
\end{figure}

\newpage
```

The result of all this code is that two pages are created with 2 figures (prediction and test_prediction.png) on the first page and 1 figure (loss-training.png) on the second.

Transfer the report.tex file (together with everything else in the experiments folder) to the results folder by using

```
ac prep sine
```

Generate the report by running on the command line

```
ac report sine/1
```

This command stores the generated report in `/home/caffe/reports/`. Find the pdf file in that folder and transfer it to your host machine to view the report. You will see a table of contents with one entry – the sine wave experiment. As you might expect, the automated reporting gets much more

useful when many experiments are run and you need to look at hundreds of plots.

We have gone through the entire workflow from model creation to training and then plotting and compiling the results into a pdf report. For this problem, the dataset was provided, but in the take-home Lab, you will be guided through making the HDF5 files by yourself. This is the basic procedure that is used for training NNs, and you will get very familiar with it by the end of the workshop.

Take-Home Lab 2B:
The purpose of this take-home lab is to get you very familiar with putting datasets into HDF5 format, which can be easily used by Caffe and other frameworks. This is needed when tackling real datasets for the Datathon.

The goal of this lab is to construct Mean, Min, Max of weekly temperature and rainfall data from raw textual datasets. This lab requires a basic familiarity with Python.

Instructions

The files required for this lab are provided in the `/home/terra/caffe/datathon-files/lab2b` directory. There are three csv files containing environmental data:

- `rainfall_total.csv`: daily total rainfall readings (23 stations)
- `temperature_max.csv`: daily max temperature readings (4 stations)
- `temperature_mean.csv`: daily mean temperature readings (4 stations)
- `temperature_min.csv`: daily minimum temperature readings (4 stations)

These stations were chosen as they had the most complete data (least missing values). Your task is to convert these data into weekly values, and then prepare them into a train and test dataset in HDF5 format.

Convert to weekly values
Open the csv files and look at the column headers. The first three columns give the year, month and day and the rest of the columns are the readings from the various stations.

Notice that there are missing values in the data, a common problem when dealing with raw data. Fortunately python has a module called pandas that handles this for you easily, however you have to be aware of the presence of missing data and their effects on the data processing that you perform.

The python code for processing the rainfall data is provided in `lab2b_1.py` and used as an example. We'll explain the important parts of the code.

The first few lines imports the relevant modules. Note the import of pandas. The `date` function from the `datetime` module converts a year, month and day value into a date object.

Two helper functions are defined.

- `ISO_date` takes in a (year, month, date) tuple or pandas series and

outputs the ISO year and week number.

- `handle_date` handles the date conversion and filtering for you. It takes a pandas dataframe as input and uses the first 3 columns of the dataframe as the (year, month, day) values to convert to (year, week number) columns. Finally it sets the year and week number as the index of the dataframe, and filters out data from before year 2000 (the last week of 1999 extends into the first few days of year 2000).

```
r_daily = pd.read_csv('/path/to/rainfall_total.csv')
r_daily = handle_date(r_daily)
```

These lines loads the rainfall data into a pandas dataframe and applies the `handle_date` function on it.

Now we start aggregating the data. First the data is aggregated across stations by row (`axis=1`):

```
r_daily_mean = r_daily.agg('mean', axis=1)
r_daily_max = r_daily.agg('max', axis=1)
r_daily_min = r_daily.agg('min', axis=1)
```

In this step pandas, automatically takes into account missing values when calculating the mean.

Next the daily mean, max and min columns are combined into a single dataframe and the columns renamed:

```
r_daily = pd.concat([r_daily_mean, r_daily_max,
                             r_daily_min], axis=1)
r_daily.columns = ['mean daily rainfall', 'max daily rainfall',
                           'min daily rainfall']
```

The data within the same week are then aggregated, with a different aggregation operation specified for each column (eg ‘mean’ operation for the ‘mean daily rainfall’ column):

```
rainfall = r_daily.groupby(['Year','Week']).agg(
                           {'mean daily rainfall':'mean',
                            'max daily rainfall':'max',
                            'min daily rainfall':'min'})
```

The final result is a dataframe consisting of week numbers as the rows and the mean, max and min data as columns. Try to do this for the temperature data.

The main change that has to be made is to load the 3 csv files instead of one and apply the appropriate aggregation operation on each.

Tip: to copy and paste a line in Vim, go to the command mode (press `Esc`), go to the line you want to copy and press `yy` , then use `p` at where you want to paste it.

Once you're done, run the python script from the command line by using

```
python <script name>.py
```

Create datasets

The h5py module allows python to read and write hdf5 files. Example code is provided in `lab2b_2.py`. Important lines are highlighted below.

Note the import of h5py module:

```
import h5py
```

The csv files are read with the first 2 columns assigned as the index.

```
temperature = pd.read_csv('/path/to/temperature_weekly.csv',
                          index_col=[0,1])
rainfall = pd.read_csv('/path/to/rainfall_weekly.csv',
                          index_col=[0,1])
```

After combining the temperature and rainfall data together in the `data` dataframe, we split it into two to form the train and test datasets. We use a split ratio of 80% - 20% for the train and test datasets, but there's no strict rule on how exactly to split it. The following lines find the boundary between train and test, and then splits the dataset accordingly.

```
train_set_length = round(0.8 * len(data))

train_data = data[:train_set_length]
test_data = data[train_set_length:]
```

To write a hdf5 file, use the `h5py.File` function and give it the path to the file you want to create and the mode (`w` means: create file, truncate if file exists). The string `'data'` in `f['data']` on the next line is the name of the dataset in the hdf5 file (ie the data in `train_data` is stored as the dataset `data` in `train.h5`)

```
with h5py.File('/path/to/train.h5','w') as f:
     f['data'] = train_data
```

In this exercise, only the data dataset (the input data to the NN) is written to the hdf5 files. However in a typical NN training run, there will also be a label dataset, so there will be an additional line when writing to the hdf5 file:

```
with h5py.File('/path/to/train.h5','w') as f:
     f['data'] = train_data
     f['label'] = train_label
```

NOTICE ON DATA SOURCES

The HDF5 and CSV data files in this and subsequent Labs contain information from Temperature and/or Rainfall and/or Dengue Case history accessed on November 2017 from Singapore's Data.gov, Meteorological Service of Singapore and Singapore's Ministry of Health websites, which is made available under the terms of the Singapore Open Data Licence version 1.0 (https://data.gov.sg/open-data-licence).

Our use of these data in this Datathon should in **no way** be construed to imply any endorsement by any Singapore Government agency.